1801210687-向建宇-第二次作业重交

题一：用letter-recognition数据集测试knn算法

**算法原理：**

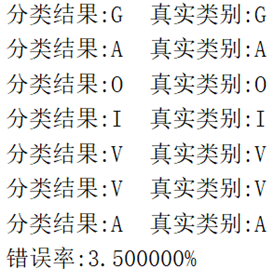
先给定训练集，即包含N个类的一些数据，然后输入测试集，每加入一个，依次与所有的训练集计算距离，取最短距离的前K训练数据，然后其中类最多的即为预测类，然后与测试集原类比较，分析错误率

**思路分析：**

首先要确定测试集和训练集的比例，和K值的大小，可以不断调整以取得最佳参数，所以步骤如下：

1. 读入数据集，并按比例划分为训练集和测试集
2. 代入公式计算dist = np.sum((inx - dataset)\*\*2, axis=1)\*\*0.5
3. 取距离最短的前K个值，并取这K个值中最多的那一个类作为预测类
4. 统计错误率

**结果截图：**



说明：我是直接取得前10%作为测试集，后90%训练集，正确率在96.5%，代码见附录1

题二：用page-blocks.data数据集测试决策树算法

**算法原理：**

先给定训练集，然后通过信息熵算出结点的前后顺序，每一个结点连接N个下一属性的结点，直到出现叶子结点，并记录当前叶子的类，这样重复遍历所有训练集就得到了一颗树，进一步优化，通过计算概率，可以用先剪枝和后剪枝，将树变小一点，最后当测试集顺着根结点到达此叶子结点时，即判断为当前叶子结点的类 ，并统计正确率。

**思路分析：**

根据树上和ppt上的算法描述可知决策树的生成算法为：

输入: 训练集D={(𝑥\_1, 𝑦\_1), (𝑥\_2, 𝑦\_2), ... , (𝑥\_𝑚, 𝑦\_𝑚),};

属性集A={𝑎\_1, 𝑎\_2, ... , 𝑎\_𝑑}

过程: 函数TreeGenerate(D, A)

1:生成结点node;

2: if D中样本全属于同一类别C then

3: 将node标记为C类叶结点; return

4: end if

5: if A = ∅ OR D中样本在A上取值相同 then

6: 将node标记为叶结点，其类别标记为D中样本数最多的类; return

7: end if

8: 从A中选择最优划分属性𝑎\_∗;

9: for 𝑎\_∗ 的每一个值 𝑎\_∗^𝑣 do

10: 为node生成一个分支; 令𝐷\_𝑣表示D中在𝑎\_∗上取值为𝑎\_∗^𝑣的样本子集;

11: if 𝐷\_𝑣 为空 then

12: 将分支结点标记为叶结点，其类别标记为D中样本最多的类; return

13: else

14: 以TreeGenerate(𝐷\_𝑣, A \ {𝑎\_∗})为分支结点

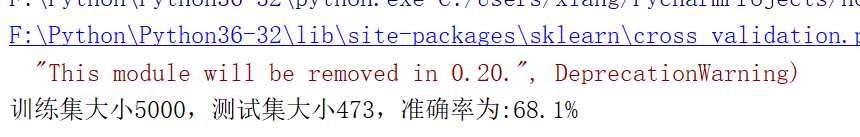
15: end if

16: end for

输出:以node为根结点的一棵决策树

注意的是：由于此次作业给的属性为连续值，所以得通过取一个中间枝来划分树枝得走向，即<中间值得向左，大于等于向右，然后将得到的决策树进行剪枝，再输入测试集进行测试。

**结果截图：**



说明：我是直接取得前473个数据作为测试集，后5000个作为训练集，正确率在68.1%，代码见附录2

附录1

|  |
| --- |
| *# -\*- coding: UTF-8 -\*-* **from** matplotlib.font\_manager **import** FontProperties **import** matplotlib.lines **as** mlines **import** matplotlib.pyplot **as** plt **import** numpy **as** np **import** operator **import** collections  **''' 函数说明:打开并解析文件，对数据进行分类  Parameters:  filename - 文件名 Returns:  returnMat - 特征矩阵  classLabelVector - 分类Label向量  Modify:  2017-03-24 ''' def** file2matrix(filename): *#打开文件,此次应指定编码* fr = open(filename,**'r'**,encoding = **'utf-8'**) *#读取文件所有内容* arrayOLines = fr.readlines() *#针对有BOM的UTF-8文本，应该去掉BOM，否则后面会引发错误。* arrayOLines[0]=arrayOLines[0].lstrip(**'\ufeff'**) *#得到文件行数* numberOfLines = len(arrayOLines) *#返回的NumPy矩阵,解析完成的数据:numberOfLines行,16列* returnMat = np.zeros((numberOfLines,16)) *#返回的分类标签向量* classLabelVector = [] *#行的索引值* index = 0  **for** line **in** arrayOLines: *#s.strip(rm)，当rm空时,默认删除空白符(包括'\n','\r','\t',' ')* line = line.strip() *#使用s.split(str="",num=string,cout(str))将字符串根据','分隔符进行切片。* listFromLine = line.split(**','**) *#将数据前三列提取出来,存放到returnMat的NumPy矩阵中,也就是特征矩阵* returnMat[index,:] = listFromLine[1:17] *#类别* classLabelVector.append(listFromLine[0])  index += 1  **return** returnMat, classLabelVector  **""" 函数说明:对数据进行归一化  Parameters:  dataSet - 特征矩阵 Returns:  normDataSet - 归一化后的特征矩阵  ranges - 数据范围  minVals - 数据最小值  Modify:  2017-03-24 """ def** autoNorm(dataSet):  *#获得数据的最小值* minVals = dataSet.min(0)  maxVals = dataSet.max(0)  *#最大值和最小值的范围* ranges = maxVals - minVals  *#shape(dataSet)返回dataSet的矩阵行列数* normDataSet = np.zeros(np.shape(dataSet))  *#返回dataSet的行数* m = dataSet.shape[0]  *#原始值减去最小值* normDataSet = dataSet - np.tile(minVals, (m, 1))  *#除以最大和最小值的差,得到归一化数据* normDataSet = normDataSet / np.tile(ranges, (m, 1))  *#返回归一化数据结果,数据范围,最小值* **return** normDataSet, ranges, minVals  **""" 函数说明:kNN算法,分类器  Parameters:  inX - 用于分类的数据(测试集)  dataSet - 用于训练的数据(训练集)  labes - 分类标签  k - kNN算法参数,选择距离最小的k个点 Returns:  sortedClassCount[0][0] - 分类结果  Modify:  2017-11-09 by Cugtyt   \* GitHub(https://github.com/Cugtyt)   \* Email(cugtyt@qq.com)  Use list comprehension and Counter to simplify code  2017-07-13 """ def** classify0(inx, dataset, labels, k):  *# 计算距离* dist = np.sum((inx - dataset)\*\*2, axis=1)\*\*0.5  *# k个最近的标签* k\_labels = [labels[index] **for** index **in** dist.argsort()[0 : k]]  *# 出现次数最多的标签即为最终类别* label = collections.Counter(k\_labels).most\_common(1)[0][0]  **return** label  **""" 函数说明:分类器测试函数  Parameters:  无 Returns:  normDataSet - 归一化后的特征矩阵  ranges - 数据范围  minVals - 数据最小值  Modify:  2017-03-24 """ def** datingClassTest():  *#打开的文件名* filename = **'letter-recognition.data'** *#将返回的特征矩阵和分类向量分别存储到datingDataMat和datingLabels中* datingDataMat, datingLabels = file2matrix(filename)  *#取所有数据的百分之十* hoRatio = 0.10  *#数据归一化,返回归一化后的矩阵,数据范围,数据最小值* normMat, ranges, minVals = autoNorm(datingDataMat)  *#获得normMat的行数* m = normMat.shape[0]  *#百分之十的测试数据的个数* numTestVecs = int(m \* hoRatio)  *#分类错误计数* errorCount = 0.0   **for** i **in** range(numTestVecs):  *#前numTestVecs个数据作为测试集,后m-numTestVecs个数据作为训练集* classifierResult = classify0(normMat[i,:], normMat[numTestVecs:m,:],  datingLabels[numTestVecs:m], 4)  print(**"分类结果:%s\t真实类别:%s"** % (classifierResult, datingLabels[i]))  **if** classifierResult != datingLabels[i]:  errorCount += 1.0  print(**"错误率:%f%%"** %(errorCount/float(numTestVecs)\*100))  **""" 函数说明:main函数  Parameters:  无 Returns:  无  Modify:  2017-03-24 """ if** \_\_name\_\_ == **'\_\_main\_\_'**:  datingClassTest() |

附录2：

|  |
| --- |
| *# -\*- coding: utf-8 -\*-* **from** numpy **import** \*  **from** scipy **import** \*  **from** math **import** log  **import** operator  **from** sklearn.cross\_validation **import** train\_test\_split  *#import treePlotter* **"""  加载训练数据和测试数据，样本中没有给出  按数据集中的要求切割出5000个样本作为训练集，剩下473个样本作为测试集  使用sklearn库中的train\_test\_split来划分，每次产生的训练集都是随机的  得到的训练集和测试集是包含了标签的  """   def** load\_data(train\_size):  data = open(**'page-blocks.data'**).readlines()   data\_set = []   **for** line **in** data:   format\_line = line.strip().split()   data\_set.append(format\_line)    data\_size = len(data)   test\_data\_size = data\_size - train\_size   train\_data, test\_data = train\_test\_split(data\_set, test\_size=test\_data\_size / data\_size) *# 测试集所占的比例* **return** train\_data, test\_data   **"""  决策树的生成，data\_set为训练集，attribute\_label为属性名列表  决策树用字典结构表示，递归的生成  """   def** generate\_decision\_tree(data\_set, attribute\_label):  label\_list = [entry[-1] **for** entry **in** data\_set]   **if** label\_list.count(label\_list[0]) == len(label\_list): *# 如果所有的数据都属于同一个类别，则返回该类别* **return** label\_list[0]   **if** len(data\_set[0]) == 1: *# 如果数据没有属性值数据，则返回该其中出现最多的类别作为分类* **return** most\_voted\_attribute(label\_list)   best\_attribute\_index, best\_split\_point = attribute\_selection\_method(data\_set)   best\_attribute = attribute\_label[best\_attribute\_index]   decision\_tree = {best\_attribute: {}}   **del** (attribute\_label[best\_attribute\_index]) *# 找到最佳划分属性后需要将其从属性名列表中删除* **"""   如果best\_split\_point为空，说明此时最佳划分属性的类型为离散值，否则为连续值   """   if** best\_split\_point == **None**:   attribute\_list = [entry[best\_attribute\_index] **for** entry **in** data\_set]   attribute\_set = set(attribute\_list)   **for** attribute **in** attribute\_set: *# 属性的各个值* sub\_labels = attribute\_label[:]   decision\_tree[best\_attribute][attribute] = generate\_decision\_tree(   split\_data\_set(data\_set, best\_attribute\_index, attribute, continuous=**False**), sub\_labels)   **else**:   **"""   最佳划分属性类型为连续值，此时计算出的最佳划分点将数据集一分为二，划分字段取名为<=和>   """** sub\_labels = attribute\_label[:]   decision\_tree[best\_attribute][**"<="** + str(best\_split\_point)] = generate\_decision\_tree(   split\_data\_set(data\_set, best\_attribute\_index, best\_split\_point, **True**, 0), sub\_labels)   sub\_labels = attribute\_label[:]   decision\_tree[best\_attribute][**">"** + str(best\_split\_point)] = generate\_decision\_tree(   split\_data\_set(data\_set, best\_attribute\_index, best\_split\_point, **True**, 1), sub\_labels)   **return** decision\_tree   **"""  通过信息增益比来计算最佳划分属性  属性分为离散值和连续值两种情况，分别对两种情况进行相应计算  """   def** attribute\_selection\_method(data\_set):  num\_attributes = len(data\_set[0]) - 1 *# 属性的个数，减1是因为去掉了标签* info\_D = calc\_info\_D(data\_set) *# 香农熵* max\_grian\_rate = 0.0 *# 最大信息增益比* best\_attribute\_index = -1   best\_split\_point = **None** continuous = **False   for** i **in** range(num\_attributes):   attribute\_list = [entry[i] **for** entry **in** data\_set] *# 求属性列表，此时为连续值* info\_A\_D = 0.0 *# 特征A对数据集D的信息增益* split\_info\_D = 0.0 *# 数据集D关于特征A的值的熵   #if attribute\_list[0] not in set(['M', 'F', 'I']): #判断离散值  #continuous = True* **"""   属性为连续值，先对该属性下的所有离散值进行排序   然后每相邻的两个值之间的中点作为划分点计算信息增益比，对应最大增益比的划分点为最佳划分点   由于可能多个连续值可能相同，所以通过set只保留其中一个值   """   if** continuous == **True**:   attribute\_list = sort(attribute\_list)   temp\_set = set(attribute\_list) *# 通过set来剔除相同的值* attribute\_list = [attr **for** attr **in** temp\_set]   split\_points = []   **for** index **in** range(len(attribute\_list) - 1):  *# 求出各个划分点* split\_points.append((float(attribute\_list[index]) + float(attribute\_list[index + 1])) / 2)   **for** split\_point **in** split\_points: *# 对划分点进行遍历* info\_A\_D = 0.0   split\_info\_D = 0.0   **for** part **in** range(2): *# 最佳划分点将数据一分为二，因此循环2次即可得到两段数据* sub\_data\_set = split\_data\_set(data\_set, i, split\_point, **True**, part)   prob = len(sub\_data\_set) / float(len(data\_set))   info\_A\_D += prob \* calc\_info\_D(sub\_data\_set)   split\_info\_D -= prob \* log(prob, 2)   **if** split\_info\_D == 0:  split\_info\_D += 1   **"""   由于关于属性A的熵split\_info\_D可能为0，因此需要特殊处理   常用的做法是把求所有属性熵的平均，为了方便，此处直接加1   """** grian\_rate = (info\_D - info\_A\_D) / split\_info\_D *# 计算信息增益比* **if** grian\_rate > max\_grian\_rate:  max\_grian\_rate = grian\_rate   best\_split\_point = split\_point   best\_attribute\_index = i   print([best\_attribute\_index, best\_split\_point])   **else**: *# 划分属性为离散值* attribute\_list = [entry[i] **for** entry **in** data\_set] *# 求属性列表* attribute\_set = set(attribute\_list)   **for** attribute **in** attribute\_set: *# 对每个属性进行遍历* sub\_data\_set = split\_data\_set(data\_set, i, attribute, **False**)   prob = len(sub\_data\_set) / float(len(data\_set))   info\_A\_D += prob \* calc\_info\_D(sub\_data\_set)   split\_info\_D -= prob \* log(prob, 2)   **if** split\_info\_D == 0:  split\_info\_D += 1   grian\_rate = (info\_D - info\_A\_D) / split\_info\_D *# 计算信息增益比* **if** grian\_rate > max\_grian\_rate:  max\_grian\_rate = grian\_rate   *# print(max\_grian\_rate)* best\_attribute\_index = i   best\_split\_point = **None** *# 如果最佳属性是离散值，此处将分割点置为空留作判定* **return** best\_attribute\_index, best\_split\_point   **"""  多数表决：返回标签列表中数量最大的类  """   def** most\_voted\_attribute(label\_list):  label\_nums = {}   **for** label **in** label\_list:   **if** label **in** label\_nums.keys():   label\_nums[label] += 1   **else**:   label\_nums[label] = 1   sorted\_label\_nums = sorted(label\_nums.items(), key=operator.itemgetter(1), reverse=**True**)   **return** sorted\_label\_nums[0][0]   **"""  计算数据集D的香农熵  """   def** calc\_info\_D(data\_set):  num\_entries = len(data\_set)   label\_nums = {} *# 为每个类别建立字典，value为对应该类别的数目* **for** entry **in** data\_set:   label = entry[-1]   **if** label **in** label\_nums.keys():   label\_nums[label] += 1   **else**:   label\_nums[label] = 1   info\_D = 0.0   **for** label **in** label\_nums.keys():  prob = float(label\_nums[label]) / num\_entries   info\_D -= prob \* log(prob, 2)   **return** info\_D   **"""  按属性划分子数据集，分为离散属性的划分与连续属性的划分  index为划分属性的下标，value在离散属性划分的情况下为划分属性的值，continuous决定了是离散还是连续属性划分  part在连续属性划分时使用，为0时表示得到划分点左边的数据集，1时表示得到划分点右边的数据集  """   def** split\_data\_set(data\_set, index, value, continuous, part=0):  res\_data\_set = []   **if** continuous == **True**: *# 划分的属性为连续值* **for** entry **in** data\_set:   **if** part == 0 **and** float(entry[index]) <= value: *# 求划分点左侧的数据集* reduced\_entry = entry[:index]   reduced\_entry.extend(entry[index + 1:]) *# 划分后去除数据中第index列的值* res\_data\_set.append(reduced\_entry)   **if** part == 1 **and** float(entry[index]) > value: *# 求划分点右侧的数据集* reduced\_entry = entry[:index]   reduced\_entry.extend(entry[index + 1:])   res\_data\_set.append(reduced\_entry)     **else**: *# 划分的属性为离散值* **for** entry **in** data\_set:   **if** entry[index] == value: *# 按数据集中第index列的值等于value的分数据集* reduced\_entry = entry[:index]   reduced\_entry.extend(entry[index + 1:]) *# 划分后去除数据中第index列的值* res\_data\_set.append(reduced\_entry)   **return** res\_data\_set   **"""  对一项测试数据进行预测，通过递归来预测该项数据的标签  decision\_tree:字典结构的决策树  attribute\_labels:数据的属性名列表  one\_test\_data：预测的一项测试数据  """   def** decision\_tree\_predict(decision\_tree, attribute\_labels, one\_test\_data):  first\_key = list(decision\_tree.keys())[0]   second\_dic = decision\_tree[first\_key]   attribute\_index = attribute\_labels.index(first\_key)   res\_label = **None   for** key **in** second\_dic.keys(): *# 属性分连续值和离散值，连续值对应<=和>两种情况* **if** key[0] == **'<'**:   value = float(key[2:])   **if** float(one\_test\_data[attribute\_index]) <= value:   **if** type(second\_dic[key]).\_\_name\_\_ == **'dict'**:   res\_label = decision\_tree\_predict(second\_dic[key], attribute\_labels, one\_test\_data)   **else**:   res\_label = second\_dic[key]   **elif** key[0] == **'>'**:   *# print(key[1:])* value = float(key[1:])   **if** float(one\_test\_data[attribute\_index]) > value:   **if** type(second\_dic[key]).\_\_name\_\_ == **'dict'**:   res\_label = decision\_tree\_predict(second\_dic[key], attribute\_labels, one\_test\_data)   **else**:   res\_label = second\_dic[key]     **else**:   **if** one\_test\_data[attribute\_index] == key:   **if** type(second\_dic[key]).\_\_name\_\_ == **'dict'**:   res\_label = decision\_tree\_predict(second\_dic[key], attribute\_labels, one\_test\_data)   **else**:   res\_label = second\_dic[key]   **return** res\_label   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   train\_size = 5000 *# 训练集大小，数据集中总共有5473项数据* train\_data, test\_data = load\_data(train\_size)   attribute\_label = [**'HEIGHT'**,**'LENGTH'**,**'AREA'**,**'ECCEN'**,**'P\_BLACK'**,**'P\_AND'**,**'MEAN\_TR'**,**'BLACKPIX'**,**'BLACKAND'**,**'WB\_TRANS'**]   decision\_tree = generate\_decision\_tree(train\_data, attribute\_label)   *# 递归会改变attribute\_label的值，此处再传一次* attribute\_label = [**'HEIGHT'**,**'LENGTH'**,**'AREA'**,**'ECCEN'**,**'P\_BLACK'**,**'P\_AND'**,**'MEAN\_TR'**,**'BLACKPIX'**,**'BLACKAND'**,**'WB\_TRANS'**]   count = 0   *# 计算准确率* **for** one\_test\_data **in** test\_data:   **if** decision\_tree\_predict(decision\_tree, attribute\_label, one\_test\_data) == one\_test\_data[-1]:  count += 1   accuracy = count / len(test\_data)   print(**'训练集大小%d，测试集大小%d，准确率为:%.1f%%'** % (train\_size, len(test\_data), 100 \* accuracy))   *#treePlotter.createPlot(decision\_tree)* |